AllGo Coding Guidelines

AllGo Embedded Systems Pvt Ltd

Version History

Version Number	Date	Author(s)	Remarks
0.1	24th Feb 2014	Kunal S	Initial draft
0.2	4th March 2014	Kunal S	Updated heading names
0.3	31st March 2014	Kunal S	Updated coding guidelines section with more data
0.4	10th April 2014	Kunal S	0.3 review sections added. Paragraphs or case studies moved to boxes.
0.5	30th April 2014	Aajna K	Updated with more sections
0.6	30th April 2014	Kunal	Added static code analysis
0.7	4th May 2014	Manjari S	Updated coding guidelines with more sections
0.8	12th May 2014	Vaisakh N	Updated with review check list and minor updates
0.9	15th May 2014	Kunal S	Accepted all review comments and updates
1.0	2nd June 2014	Kunal S	Reformatted font size and cleanup

Contents

1. Introduction

The purpose of this document is to describe and explain various practises, guidelines and conventions which should be followed to improve overall code quality and readiablity.

2. Acronyms and Abbreviations

OSAL	Operating System Abstraction Layer
API	Application Programming Interface

3. Ready to code ?

Before starting with the coding phase, there needs to be a clear entry point.

Final design of the module to be implemented should be ready, derived from the list of requirements.

The design can be in the form of code files (source and header files) defining only API function prototypes with initial comments.

1. Design interface with respect to code files

- Header files
- internal .h files
- external .h files (Public API)
- Source files split / structure
- internal source files (Core API)
- external source files (Public API)
- Function stubs defined with ONLY initial comments.
- Whole design flow can be put as file or function header comment.

4. What is expected ?

There are two important sides of coding, which developer should improve.

4.1. Technical side

The technical side consists of writing a piece of code considering those machines, on which they need to be executed.

You should be developing your program which needs to be better in quality and robustness, specific to your machine.

For improving on the technical side following points should be improved:

- Functionality
- Efficient code
- Robustness
- Optimized

4.2. Presentation

Coding presentation consists of writing a piece of code considering they need to be readable and understandable by humans.

Following important points which gives your code good presentation:

- Readability - Look and Feel
- Use of Tabs, indents, notations, num of lines in a function
- Using standard naming conventions
- Maintainability (Use of simple algorithm)
- Properly commented

5. Collection of resources

The resource consists of collection of motivational / good practices of coding which will help as a guide to improve both coding quality and readability.

The resource will be in the form of document or sheet maintained on Gerrit. The document can be specific to coding language or common practices which can be followed as general coding practice.

Developers can contribute and update the documents which can be reviewed and added to the list.

Upload and put links here :

6. Lets meet before coding

Precoding meeting involves presentation of a standard coding guidelines document before every project kickoff.

This document should be maintained and updated at repository regularly.

Document content:

1. Standard conventions
2. Guidelines
3. File structure
4. Topics specific to a project

- Re-entrant functions
- Constraints on stack/global etc.
- Specific conventions
- Team meeting

Debugging strategy:

1. Logs
2. Debug levels
3. Unit testing
4. Compilation flags in Makefile (warnings , language checking)
5. Debug tools (lint, valgrind static analysis of code)

7. Coding presentation

Let's look out different methods to improve your code readability and better understanding.

7.1. Naming your code

It is important to follow a good naming convention for the symbols in your programs.

This is especially important for libraries, since they should not pollute the global namespace. It is very annoying when a library has sloppily-named symbols that clash with names you may want to use in your programs.

7.1.1. File names

- File names are lowercase, with underscores to separate words in the file name.
- File names can be in the form of ***module_submodule_operation***.* where * represents c, h extensions.

7.1.2. Function names

- Function names are lowercase, with underscores to separate words.
- Function names can be in the form of ***module_submodule_operation()***.
- Names should be as descriptive as possible, but keeping in mind that it need not be over-descriptive.
- Functions which are local to the sub-module only, i.e. the functions inside the module which are not called from other library or application, should follow the naming convention given below:

    1. Function names are lowercase, with underscores to separate words.
    2. Function names can be in the for of: ***submodule_operationcategory_subcategory().***

7.1.3. Function Calls

Functions should be called with no spaces between the function name, the opening parenthesis, and the first parameter; spaces between commas and each parameter, and no space between the last parameter, the closing parenthesis, and the semicolon.

7.1.4. Variable names

- Types and structure names can be in mixed upper and lowercase.
  For example: BrowsingScope, EntryList.
- No use of underscore in the variable names.
- Names should be as descriptive as possible, but keeping in mind that it need not be over-descriptive. This makes code very easy to read and almost self-documenting.
- First letter, i.e. data type prefix in small letter and subsequent words starting with first letter in capital. For example: ***pArtistType, gFolderIndex***
- Prepend each variable name with the variable type:

CHAR	"c"
UCHAR	"uc"
BYTE	"uc"
INT16	"w"
UINT16	"uw"
INT32	"i"
UINT32	"ui"
INT64	"l"
UINT64	"ul"
BOOL	"b"
Structure	"s"
Enum	"e"
void	"v"
Array	prepend "a" eg. auc
Pointer	prepend "p" eg. pui
Pointer to pointer	prepend "pp" eg. ppui
Global	prepend "g" eg. g_Device
Union	prepend “un”

Function pointer prepend “fp”

- In case of temporary variables like loop index, avoid using descriptive names or prefixes, only in this case it is recommended to use i, j, etc.

7.1.5. Constants

- Constants should always be all-uppercase, with underscores to separate words.
- Module-defined constant names should also be prefixed by an uppercase spelling of the module that defines them.
- For Enum constants its recommended to prepend the constant with 'E_”

7.2. Formatting code

Visual appeareance, readability, reuse plays a vital role in coding.

Code fomatting helps you acheive this. A formated code helps in better understanding of the code by the peers who will use the same code sooner or later. Insome cases it avoids inducing errors as well.

7.2.1. Always indent

- Limit the length of source line to 80 characters. This enhances the readability.
- Always indent your code. This gives a better visual appearance. There are many short cuts in various editing tools like gvim which will automatically indent your code.
- Always use space in place of tabs. This helps keep the code view uniform across different editors.
- "If" blocks should always be enclosed with braces starting on the next line of the if condition even if the block is of only one line.

7.2.2. Control Structures

- Control statements should have one space between the control keyword and opening parenthesis, to distinguish them from function calls.
- Split long control statements into several lines when the character/line limit would be exceeded.

7.2.3. How to put comments

- Comments help reader as well as the coder understand what he/she is doing. It must be present wherever required.

  For Example, there is no need to add comment when you do operations like a+b saying adding a and b, but when you do a complex operation, say iHeaderFrame*2 + iOffset/5, then the comment should be such that it explains why iHeaderFrame is being multiplied by 2 and why iOffset is being divided by 5.
- Always add your comments between /**/ .

7.2.4. Using doxygen style

- Doxygen is an open source document generator tool. This acts as a cross reference documentation and code.

- The generic syntax of documentation comments is to start a comment with an extra asterisk after the leading comment delimiter.

  Eg:

```
/**
<A short one line description>

<Longer description>
<May span multiple lines or paragraphs as needed>

@param  Description of method's or function's input parameter
@param  ...
@return Description of the return value
*/
```

- Doxygen offers options to group functions/data structures into different modules. These increase the readability of the documentation generated.

- Detailed description on the various features offered by doxygen can be found at www.doxygen.org

7.3. Headers

A header file is a file containing C declarations and macro definitions to be shared between several source files.

Including a header file produces the same results as copying the header file into each source file that needs it. Such copying would be time-consuming and error-prone.

With a header file, the related declarations appear in only one place. If they need to be changed, they can be changed in one place, and programs that include the header file will automatically use the new version when recompiled.

7.3.1. Brief file content description

A normal header should first have a preprocessor directive like below:

#ifndef HEADER_FILE

#define HEADER_FILE

Header file content

#endif

This ensures the header is included only once.

NOTE : Make sure no two different headers have the same #ifndef. Otherwise the second file contents will not be included and you will be looking at undefined errors .

Next is to include all the preprocessor directives like, #define PI_VALUE 3.14.

Remember to add comments to the directives as well. There can be pre-processor directives which define the buffer size or number of retries. A small comment there on why this value is chosen will help in a large way during code maintainence.

After this you can define your enums followed by structures and at the end your function prototypes.

Every member of the enum and structure should be documented in doxygen style. There should also be a brief description for the structure or enum itself indicating its usage.

The function declarations should be documented in detail describing its functionality, parameters, return values, pre conditions.

7.3.2. Maintain file update history

We should make a habit of maintaining file update history.

Example,

```
*
• CHANGE HISTORY
* ---------------------------------------------- ----------------------------
• DATE          REVISION    AUTHOR      COMMENTS
* ---------------------------------------------- ----------------------------
• 01/09/2011    1.0         Employe1    Initial version
• 16/09/2011    1.1         Employe2    Updated with new design details.
*
```

7.3.3. C++ compilation support

```
#if defined __cplusplus

extern “C” {

#endif
```

Then, the following lines should be at the bottom of the header file to logically close the #ifdef as added earlier.

```
#if defined __cplusplus

   }

#endif
```

Adding these lines in the header files gives provision to compile the C library with g++ compiler also.

8. Coding Guidelines

One of the main aim of coding guidelines is to keep *code complexity* as low as possible. *Complex* means hard to read or error prone.

8.1. Avoid nesting

C allows to chain and nest expressions, but it becomes very difficult to keep track of how many condition you have put and for what purpose.

Some times it is quite necessary to use but most of the time it could be avoided as instead of nested condition,we could use multiple condition if at all we already know those are mutually dependent events.

This can evoke following problems:

1. Code can become unreadable.
2. Source code debugging doesn't show intermediate data.
3. Expressions need to be wrapped to a second line or the line will be extraordinary long.

Most of these points make code harder to read and debug.

Cyclomatic complexity

Cyclomatic complexity is a measure of how complex a function is, based on the number of 'decision points' in the function. The decision points are 'if', 'while', 'for', and 'case labels. The higher the number, the more complex the function.

It is considered bad to have your complexity values too high is because it makes the function difficult to test. In order to test a function to its full potential, you need to have a separate test for each possible code path.

The number of code paths increases exponentially with every new decision point, which means that by the time you've got more than a handful of decisions in a single function, you start needing hundreds of tests in order to be sure you've covered the whole range of functionality it might perform.
Hundreds of tests for a single function is clearly too many, so the better option is to reduce the number of decision points per function by splitting it into several smaller functions with fewer decisions each.

8.2. Don't copy same code

Code fragments should not be repeated.

Redundant code is harder to maintain and increases the probability of introducing defects. Code with many redundancies is harder to read.

You might consider implementing a more general function instead.

If performance is the problem, use macros or an optimizing compiler that inlines the functions.

Case study:

I've encountered this situation with one of the colleague of ours, well she was using a segment of code by copying it from some other segment ..and since the other segment was working fine she safely assumed that it would work also .. But after integrating when she gave it to the customer for testing .. they found some bizarre behavior. How? Well the same code was working fine with some other module but not on this one how ?

Well code snippet was containing one instruction like :

auiArrayToSend[MAX_SIZE] = 20;
where MAX_SIZE is the size of the Array .

Well once she started using it in her segment of code. It was corrupting some part of the process address space which was getting used, but in the earlier segment even though it was corrupting the address space of some process it was not visible since it was not getiing used. So as far as, copying the code is concerned, don't do it, but if you are doing then just cross verify each and every instruction that you might have copied.

8.3. Remember to initialize all

Write deterministic code. Stack variables and buffers should be initialized.

You should consider to assign a freed pointer as NULL.

Example,

```
int main ()
{
    int count;
   while(count<100)
   {
    printf (“%d”,count);
    count++;
   }
}
```

8.4. Build in modules

Modularity is an important key for code maintainability and handling complexity.

If you introduce a *module abstraction layer* on top of the functions, you can decrease problem complexity tremendously.

While building modules, it is of utmost importance that we keep in mind the packaging criteria.

1. What all header files will be part of core library and which all will be exposed to application as part of API functions ?

2. What all object files should be made collectively a shared library or static library ?

3. How these libraries will be used to generate binaries ?

   We need to keep in mind the customers requirements, what part of the module they want as library and what should be generated as binaries.

8.5. Passing function Parameters

Functions can have multiple parameters. When there is a huge list of parameters, it is better to form a structure and pass a pointer to that structure as an argument to the function. It saves the stack space.

This becomes significant when working on hardware with limited RAM.

8.6. Keep functions in order

There are two traditional styles of code order (i.e. Order of function calls in a source file).

Pascal style defines a function before it is referenced. Using Pascal style makes it unnecessary to track function interface changes in the prototypes.

C style defines a function after it is referenced. C style sources seem to be easier to read since going from the start of the file to the end is much more like the actual program flow compared to Pascal style.

It increases the readability and also it improves the modularity of the program. Even in a single file the set of functions should be categorized in such a way that it should be very easy to jump to any section of the entire file.

Error handling should typically be done without delay, e.g.

Fp = fopen(file, "r"); if(fp == 0) return -1; process(fp);

is preferable to

fp = fopen(file, "r"); if(fp != 0) process(fp); else return -1;

It is more readable since it doesn't add one layer of indentation per condition and the 'process' part above may be large and tear the blocks quite apart.

8.7. Design re-entrant functions

Re-entrant function means a function which can be called from different applications at the same time.

These functions can be interrupted in the middle of its execution and then safely called again ("re-entered") before its previous invocations complete execution.

Rules of reentrancy:

1. Reentrant code may not hold any static (or global) non-constant data.
2. Reentrant code may not modify its own code.
3. Reentrant code may not call non-reentrant computer programs or routines. An example of such function is **strtok()**.

8.8. Access restrictions

The *scope* of an identifier (function, type, etc.) is the part of the code in which the identifier can be referenced.

Choosing the *scope* of a function, variable or type is one of the most important *micro architecture* instruments.

The C programming language offers (From broad to narrow) :

- Application scope
- File scope
- Function scope
- Block scope

Generally, scope should be chosen as narrow as possible.

8.9. Function's scope

Functions have file scope (static) or application scope (not static).

Limit a function to file scope if possible. This affects the *modularization* of a software component (i.e. the way functions are grouped together to source files).

We need to see the scope of the function, by encapsulating accessibility of the function. We can make sure that only relevant procedure or objects could access the function.

It should be completely invisible to set of objects which has no business to access it.

Case of C:
Carefully using the static scope of the function will tell at the compile time itself that no other file could access this function so that way enormous pressure could be reduced from the compiler.

A static function is not visible outside of its translation unit, which is the object file it is compiled into. In other words, making a function static limits its scope.
As the function is clearly marked private to the file, the compiler is in a better position to generate a complete call-graph for the function. This may result in the compiler deciding to automatically in-line the function for better performance.

All functions are implicitly declared as extern, which means they're visible across translation units. But when we use static it restricts visibility of the function to the translation unit in which it's defined.

8.10. Keeping variables alive

Variables can have all four kinds of scope: application scope, file scope, function scope, block scope. *The variable scope should be as small as possible.* The opinions about using variable declarations in block scope differ.

The use of global or static variables should come by design.

Typically, there is no overhead (stack pointer operations) involved with block scope variables, space for the deepest possible block allocation gets reserved at function entry.

Application scope leads to *global variables*, which generally should be avoided. Use functions to access the data (getX() and setX()) instead.

Application scope or file scope (as well as static data in functions) lets the functions that access the data only be useable by a *single thread*.

A reference to a calling functions buffer should be used instead of global or static data to avoid multithreading problems and buffer overwrite problems (e.g. as with localtime()).

Register is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size, so depending upon

the requirement we need to use the proper storage class for variables.

A storage class defines the scope (visibility) and life time of variables and/or functions within a C Program.

Why Global Variables Should Be Avoided When Unnecessary

Non-locality *:*
Source code is easiest to understand when the scope of its individual elements are limited. Global variables can be read or modified by any part of the program, making it difficult to remember or reason about every possible use.

No Access Control or Constraint Checking *:*
A global variable can be get or set by any part of the program, and any rules regarding its use can be easily broken or forgotten. (In other words, get/set accessors are generally preferable over direct data access, and this is even more so for global data.) By extension, the lack of access control greatly hinders achieving security in situations where you may wish to run untrusted code (such as working with 3rd party plugins).

Concurrency issues:
if globals can be accessed by multiple threads of execution, synchronization is necessary (and too-often neglected). When dynamically linking modules with globals, the composed system might not be thread-safe even if the two independent modules tested in dozens of different contexts were safe.

Namespace pollution *:*
Global names are available everywhere. You may unknowingly end up using a global when you think you are using a local (by misspelling or forgetting to declare the local) or vice versa. Also, if you ever have to link together modules that have the same global variable names, if you are lucky, you will get linking errors. If you are unlucky, the linker will simply treat all uses of the same name as the same object.

Memory allocation issues *:*
Some environments have memory allocation schemes that make allocation of globals tricky. This is especially true in languages where "constructors" have side-effects other than allocation (because, in that case, you can express unsafe situations where two globals mutually depend on one another). Also, when dynamically linking modules, it can be unclear whether different libraries have their own instances of globals or whether the globals are shared.

Testing and Confinement *:*

Source that utilizes globals is somewhat more difficult to test because one cannot readily set up a 'clean' environment between runs. More generally, source that utilizes global services of any sort (e.g. reading and writing files or databases) that aren't explicitly provided to that source is difficult to test for the same reason. For communicating systems, the ability to test system invariants may require running more than one 'copy' of a system simultaneously, which is greatly hindered by any use of shared services - including global memory - that are

not provided for sharing as part of the test.

8.11. Place types in header

Types should be limited to file scope or application scope, since it is a rare case that only one method (function) acts on a type (that is then in function scope).

Don't redefine types in file scope, use *common header files* instead.

Always keep structure definitions, enumerations, function prototypes in the header files with proper description.

8.12. When to use macros

Macros have file scope by nature.

However, macro encapsulating code fragments as

```
#undef m #define m…
...code...
#undef m
```

have been seen. They allow to hide macros from the rest of the source file. Limiting macro scope is a bit obscure.

One reason macros are used is performance.

They are a ways of eliminating function call overhead because they are always expanded in-line for function call.

We can use macros because these will be resolved in preprocessing itself, so it would reduce a lot of burden like creating a different stack frame like for a function call and lot of other processing activity with respect to a function. It will definitely save some cpu cycle.

Words of precaution on using MACROS:

Firstly if you have a macro that acts like a function then call it in a way that looks like a function. Call it with parenthesis after it's name (even if it takes no parameters) and then put a semi-colon after it. This will avoid confusion for the next programmer who works on your code.

Don't ever have a macro that has a "return" statement. There's nothing more annoying when you're trying to read the code of an unskilled programmer and you have to do a search on the contents of every second line of code to determine if it's a macro, and if so whether it does a return. If a line of code is setting variables or doing data IO I can generally get the idea of what the code is doing without reading the source of the function, with a well written macro it's the same. If the macro may be doing a return then the code is much harder to read and productivity goes down.

Macros should never use local variables in your function unless they are defined locally - even then it's usually a bad idea. If they allocate their own variables (something you don't want to do if you can avoid it) then they should start with XX or some other code that makes a name clash with macro parameters very unlikely. There are situations (such as code involving varargs) where a macro defined in the local source file which uses variables in the function can save a lot of duplicated code and save the programmer time. That's OK. But having source in a dozen modules having variables with particular names for macros to use makes for code that's difficult to maintain.

Put parenthesis around all macro parameters. If the macro expands to a numeric expression then also put parenthesis around the macro itself. See the following:

*#define MACRO(XX) XX*2*
*a = MACRO(b + 1); // expands to "a = b + 1 * 2" not "a = (b + 1) * 2"*
as we want,

define MACRO2(XX) XX +1
*c = MACRO2(d) * 2; // expands to "c = d + 1 * 2" not "c = (d + 1) * 2"*

*Here's the correct way to define those 2 macros and avoid nasty surprises: #define MACRO(XX) ((XX)*2)*
#define MACRO2(XX) ((XX)+1)

When defining a macro that expands to multiple statements then it's good to do something like the following example from the include/linux/sched.h in the source to the Linux kernel 2.1.99:

```
#define SET_LINKS(p) do { \
        (p)->next_task = &init_task; \
        (p)->prev_task = init_task.prev_task; \
        init_task.prev_task->next_task = (p); \
        init_task.prev_task = (p); \
        (p)->p_ysptr = NULL; \
        if (((p)->p_osptr = (p)->p_pptr->p_cptr) != NULL) \
                (p)->p_osptr->p_ysptr = p; \
        (p)->p_pptr->p_cptr = p; \
        } while (0)
```

Don't use multiple definitions for the same macro, let say we have used IPOD_MAX_BUFFER_SIZE 255 at some header file which has been included in some source file.
Now in some different process which actually has dependency on this MACRO we made it IPOD_MAX_BUFFER_SIZE 128 .

Now you can figure out how disastrous it can be if one process is relying on the fact that the other process is going to use the same definition. Well just consider the memcpy example where although the memory is allocated 128 but we are trying to copy 255. Crash.

8.13. Dangers in casting

Compilers do not generate errors or warnings on semantically false explicit casts. The explicit casts are accepted as is.

Use explicit casts as rarely as possible. It's good to think whether an explicit casts is necessary and what the compiler will do with it.

The C language allows implicit conversions from T* to void* and vice versa. There is no explicit cast needed in C to convert from void*. malloc() is an example of an often used function that returns void*.

Problems with casting:

C casts have a number of problems. In some cases, the cast does nothing more than tell the compiler (in essence): "shut up, I know what I'm doing" i.e., it ensures that even when you do a conversion that could cause problems, the compiler won't warn you about those potential problems.

Just for example, char a=(char)123456;. The exact result of this implementation defined (depends on the size and signedness of char), and except in rather strange situations, probably isn't useful. C casts also vary in whether they're something that happens only at compile time (i.e., you're just telling the compiler how to interpret/treat some data) or something that happens at run time (e.g., an actual conversion from double to long).

Explicit casting you tell the compiler to shut up, because you think you know better. In case you're wrong, you will usually only find out at run-time. And the problem with finding out at run-time is, that this might be at a customer's.

8.14. Know your data size

Know what your integer variable size is. Is it 16 bit, 32 bit, 64 bit, 128 bit?

int size limits the range of numbers you can use. Check if e.g. 31 bit (signed int) is enough for your *problem domain*.

As far as the ISO C standard (the official definition of the language) is concerned, accessing an array outside its bounds has "*undefined behavior*".

Case study:

In the "best" case, you'll access some piece of memory that's either owned by your currently running program (which might cause your program to misbehave), or that's not owned by your currently running program (which will probably cause your program to crash with something like a segmentation fault).

That's assuming your program is running under an operating system that attempts to protect concurrently running processes from each other. If your code is running on the "bare metal", say if it's part of an OS kernel or an embedded system, then there is no such protection; your misbehaving code is what's supposed to provide that protection. In that case, the possibilities for damage are considerably greater, including, in some cases, physical damage to the hardware.

Even in a protected OS environment, the protections aren't always 100%. There are bugs that permit unprivileged programs to obtain root access, for example. Even with ordinary user privileges, a malfunctioning program can consume excessive resources (CPU, memory, disk), possibly bringing down the entire system. A lot of malware (viruses, etc.) exploits buffer overruns to gain unauthorized access to the system.

(One historical example: I've heard that on some old systems with core memory, repeatedly accessing a single memory location in a tight loop could literally cause that chunk of memory to melt. Other possibilities include destroying a CRT display and moving the read/write head of a disk drive with the harmonic frequency of the drive cabinet, causing it to walk across a table and fall onto the floor.)

If we even accidentally corrupt any memory segment then any one of the following cases could arise :

This memory can have any value. There's no way of knowing if the data is valid based on your data type.

1. *This memory may contain sensitive information such as private keys or other user credentials.*
2. *The memory address may be invalid or protected.*
3. *The memory can have a changing value because it's being accessed by another program or thread.*
4. *Other things use memory address space, such as memory-mapped ports.*
5. *Writing data to unknown memory address can crash your program, overwrite OS memory space, and generally cause the software to implode.*

The following three cases illustrate the most common types of array-related illegal memory access:

Case A

```
/* "Array out of bounds" error valid indices for array foo are 0, 1, ... 999 */
 int foo[1000]; for (int i = 0; i <= 1000 ; i++) foo[i] = i;
```

Case B

```
/* Illegal memory access if value of n is not in the range 0, 1, ... 999 */
int n; int foo[1000]; for (int i = 0; i < n ; i++) foo[i] = i;
```

Case C

```
/* Illegal memory access because no memory is allocated for foo2 */
float *foo, *foo2; foo = (float*)malloc(1000); foo2[0] = 1.0;
```

These type of errors are almost impossible to track in big softwares, so make sure to take care of boundary condition checking while coding or else somebody else (debugger) will

have to pay a heavy price for what mistake you have committed .

8.15. Dangers of magic numbers

There are several methods to refer to an array size (sample array being int a[32];):

- 32
- SIZE(a macro used for the array definition)
- sizeof(a)/sizeof(*a)

The first one is the worst, it will be invalid if the array definition is changed without tracking the other occurrences of the size. So avoid using constants. The second one is better, but the third using *sizeof* is the preferred one.

Defining magic numbers helps:

1) Assist in maintenance.

If you had just developed your code like this: #define BUFFER_SIZE_FOR_ARRAYS 256 #define DISK_SIZE 1440

Now if we make use of these macros instead of literals then life would be much easier. Later at some point of time, if we have to change the value of these constants then we just need to replace the value in front of the constant, just changing that constant in one place without affecting all other constants with the same value.

2) Increases the readability of the program.

Short answer is that named constants make your life a lot easier. That really should be reason enough for using them. But we need to make sure not to use these sort of silly defines, #define SEVENTY_TWO 72

I'm not exactly sure what they think they're gaining so make sure to name the macros as per the uses.

8.16. Provide buffer size

It is strongly discouraged to implement or use functions that require a buffer as an argument, without also requiring the buffer size.

This rule should be strictly followed if the input size is an external (and hence uncontrolled) property (e.g. a line length with gets()).

Buffer overflows can:

- Corrupt adjacent data
- Corrupt the stack frame (if on the stack)
- Corrupt malloc internal data (if on the heap)

The first one is hard to find because it can subtly change the program logic. The last one is also hard to find, since the program often crashes at some later point in a call to malloc() or free(). In that case often only a malloc debug package helps.

As we know arrays as parameters and dynamically allocated memory has fixed set of buffer size . So keeping track of this buffer size is of utmost importance otherwise it can create scenario in which we can face Buffer overflow as well as well underflow, which can create a disastrous situation sometimes.

Avoiding Buffer Overflows and Underflows:
Buffer overflows, both on the stack and on the heap, are a major source of security vulnerabilities in C, Objective-C, and C++ code.
Every time your program solicits input (whether from a user, from a file, over a network, or by some other means), there is a potential to receive inappropriate data. For example, the input data might be longer than what you have reserved room for in memory.

When the input data is longer than will fit in the reserved space, if you do not truncate it, that data will overwrite other data in memory. When this happens, it is called a ***buffer overflow****. If the memory overwritten contained data essential to the operation of the program, this overflow causes a bug that, being intermittent, might be very hard to find. If the overwritten data includes the address of other code to be executed and the user has done this deliberately, the user can point to malicious code that your program will then execute.*

Similarly, when the input data is or appears to be shorter than the reserved space (due to erroneous assumptions, incorrect length values, or copying raw data as a C string), this is called a ***buffer underflow****. This can cause any number of problems from incorrect behavior to leaking data that is currently on the stack or heap.*

Although most programming languages check input against storage to prevent buffer overflows and underflows, C, Objective-C, and C++ do not. Because many programs link to C libraries, vulnerabilities in standard libraries can cause vulnerabilities even in programs written in "safe" languages. For this reason, even if you are confident that your code is free of buffer overflow problems, you should limit exposure by running with the least privileges possible. See "Elevating Privileges Safely" for more information on this topic.

Keep in mind that obvious forms of input, such as strings entered through dialog boxes, are not the only potential source of malicious input. For example:

1. *Buffer overflows in one operating system's help system could be caused by maliciously prepared embedded images.*
2. *A commonly-used media player failed to validate a specific type of audio files, allowing an attacker to execute arbitrary code by causing a buffer overflow with a carefully crafted audio file There are two basic categories of overflow: stack overflows and heap overflows. These are described in more detail in the sections that follow.*

8.17. The Goto Conundrum

Most programmers will agree that properly written, well-structured, code doesn't require the use of goto.

But, the problem is, if you simply state "no goto" you might as well say "no JMP command in the generated assembly code" as well. When we write and compile code, the result is assembly/machine code that the processor can execute.

In the assembly language you'll notice the JMP command all over the place, usually coupled with a test for some condition.

This is one way that branching and code flow control is performed in the CPU. Basically, it's the same as a goto statement, and if you subscribe to the "no goto" rule, why not eliminate it also?

It is recommended not use goto when your code wants to jump or loop in upwards direction.

Using it to jump in error condition from start of the function to the return is fine but nothing more than that.

Example,

```
function()
{
        /* Check input arguments */
        if(arg1 == NULL || arg2 == NULL)
        {
                /* Invalid Parameters */
                ret = INVAILD_PARAMS_ERROR; goto ErrorExit;
        }

  ErrorExit:
  /* Error */
  return ret;
}
```

8.18. Portability

Program code should **never ever** contain "hard-coded", *i.e.* literal, values referring to environmental parameters, such as absolute file paths, file names, user names, host names, IP addresses, URLs, UDP/TCP ports. Otherwise the application will not run on a host that has a different design than anticipated. Such variables should be parametrized, and configured for the hosting environment outside of the application proper (*e.g.* property files, application server, or even a database).Use OSAL functions in code for better portability

Note :

OSAL is an abstraction of the common system functionality that is offered by any Operating system by the means of providing meaningful and easy to use Wrapper functions that in turn encapsulate the system functions offered by the OS .So by using it portability of the code will improve.

8.19. Infinite loops

There will be some threads which need to keep running continuously throughout the execution. It is very important to make sure that these threads do not keep polling continuously. This would simply eat up your cpu cycles. Every thread which needs to run infinitely will definitely be waiting for some input or trigger based on which it would be processing. So make sure these operations are blocking so that the thread goes to sleep in the wait state.

8.20. Logging

Adding the correct logs at correct places is very important. Adding too many logs is not good and at the same time code with no logs is also not useful as it will be difficult to debug issues when reported . Logging levels are present in every project but adding the logs at correct places and using the correct logging level is upto the developer.

Some dos and donts for logging:

1.Critical information should always be logged

2.Debug logs should be removed once debugging has been completed

3.The logs should be short but informative

4.Too much log will only clutter the screen and in some cases even degrade the performance

5.Avoid putting logs in loops which keep executing continuously

8.21. Review the code yourself

Self review of the code helps to catch bugs which otherwise might be found only after many iterations of the testing. Review your code and see if you have :

1. Deallocated all the memory that you have allocated
2. Allocated all pointers before using them
3. Added breaks in switch statements

9. Don't ignore your warnings

Compiler warnings must not be ignored. While compiling the -Werror flag should be passed to gcc. In addition to -Wall, -Wextra enables extra warning flags which should be made part of the makefile.

Even if you've fixed every compiler error and linker error your compiler gives, it may still give some "**warnings**".

These warnings won't keep your code from compiling (unless you ask the compiler to treat warnings as errors), and you might be tempted to just ignore them. There are, however, some good reasons for not to do so.

In this section, we will try to see these reasons, and in doing so, we'll also see some examples of the types of warnings you might face, what they mean, and how to fix them.

9.1. Catch bugs before testing

The compiler warnings should be seen as indicators of future bugs that would otherwise be seen only on run time.

For example, if you see compiler warning “assignment in condition”, you had probably written something like:

```
if (x = 5)
{
   …..
}
```

while in all probability you intended to write:

```
if (x == 5)
{
   …..
}
```

Just for fun, analyze how tough it may be to find this bug at run time:

```
if (x = 0)
```

```
{
   printf (“x is ZERO\n”);
}
```

Catch bugs that are hard to find in testing/debugging

Considering another warning, “Variable may be used uninitialized”, it is likely that you had written something like:

```
void function (void)

{

   int x;

   if (5 == x)

   {

      …..

   }

}
```

In which case, the behavior is deterministic, depending on value of auto variable which by default is never initialized.

9.2. Optimize your code with compiler's help

Some of compiler warnings may not be indicating any bug or issue, but a close look at them, and fixing them, may help in some optimization.

For example, “Value to variable assigned but not used”. To understand this, let us see below example:

```
void function (void)
{
   int ret = 0;
   ret = iGetGlobalValue();
   if (0 != ret )
   {
      ...
   }
}
```

Here compiler may give you warning that ret is assigned a value that is never used. So, in a way, here you can skip initialization of your auto variable “ret”, which will save you few processor cycles.

9.3. Are there good warnings?

Having discussed above types of warnings, do you wonder if you must remove all warnings and not leave even a single one?

Hold on!! To the contrary of what what you think, there may be some warnings which are deliberate. Or in other words, you actually want that warning to be present.

Let us see an example. Of an RTOS based player, where we define run time memory section of each code segment. In a particular system, one may define same run time memory section for all decoders.

For this compiler will warn us, as expected, that more than one code segment may use same memory sections. But that is probably we wanted, right?

In a typical player, at one time, either MP3 would play, or WMA would play and so on. In other words, one decode is needed at one time, so we actually save on memory requirement by sharing code segments.

Other category of warnings that one may not wish to get rid of, is from open source headers. It is always preferable not to modify an open source file unless really needed.

For example, in RACE there is a warning that __cancel_arg_ can be used uninitialized.

But a closer look shows that the macro in pthread headers, do not initialize this particular variable, and is not used also. So it is decided to live with this warning.

9.4. Lessons to take away

While coding and compiling, our mindset should be that compiler is the best tool available to me for catching my bugs as early as possible.

If you do not understand what a compiler warning means, it is possibly the best to trust that compiler is telling you something valuable.

Other very important thing is, that often we hear the I am doing code cleanup and this code cleanup is expected to be a phase that includes many things including warning removal. Why write a dirty code to clean up later? Address the warnings as soon as you see them.

9.5. Certain things to know about GCC and warnings

Compiler warnings must not be ignored. While compiling the -Werror flag should be passed to gcc. In addition to -Wall, -Wextra enables extra warning flags which should be made part of the makefile.

Arguments for Gcc

-w Inhibits all warning messages – NEVER use this.

-Werror Makes all warnings into errors.

-Wall This enables all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning), even in conjunction with macros.

-Wextra This enables some extra warning flags that are not enabled by -Wall. (-Wempty-body, -Wignored-qualifiers, -Wmissing- field-initializers, -Wmissing-parameter-type, -Woverride-init, -Wsign-compare, -Wtype-limits, -Wuninitialized, -Wunused-parameter

-Wall, -Wextra and -Werror are recommended to be used in all projects.

10. Know your compiler flags

TODO

11. Double check your code

Usage of some code checking and debugging tools mandatory to have clean and error free implementation result.

1. Valgrind – Program analyzer / dynamic checker

Valgrind not only catches memory leaks but also tells you about invalid reads and writes. The memory leaks might not be accurate but the invalid reads and writes are always correct!! So do not ignore them. It helps resolve crashes which occur randomly.

For example, if you had an array of size 10 bytes(int a[10]) in the program and there is an assignment like

a[10] = 0x09;

Valgrind will show an invalid write message at this point :

Invalid write of size 1

2. Splint – Static checker for C programs
3. Lint

Allows:

- To catch some of K&R1 pitfalls (insufficient type checking, missing declarations)
- to check inter-module issues (inconsistent interface declarations, e.g. char p[] vs. extern char* p)
- to find globally unused functions and variables
- to find dead (unreachable) code (which is usually an error)
- to check for false or missing arguments to printf() or scanf()
- to check static array's bounds
- to catch inconsistent function use (e.g. ignoring return values sometimes)

4. Metrics

Metrics are a means to *quantify* some code characteristics. These quantifications can be used to achieve a desired level of commenting or modularization, etc.

Simple metrics count e.g. the

- number of source code lines
- number of lines of code (*LOC*, leaving out empty lines and comments)
- number of comments
- number of statements
- ratio of comments per statement

11.1. Static Code Analysis

Everytime you write code, are you sure your code works for all scenarios. Answer may be yes when its a small piece of code, but when it grow large to be a software with lot of features then answer is absolutely no. So you write all test cases you possibly think of and try to execute them and some times you just limit your testing to white box testing. By testing like this there is good possibility that some part of code you wrote remains untested. This is a saying “Code which is not tested is not trusted”. When your code goes live then there is good chance that it fails.

How to avoid this? There are muliple ways, one of which is of using code analysis tools to know possible threats in your program. There are two ways of doing code analysis, static code analysis and dynamic code analysis. In this article we discuss about static code analysis.

○ Why static code analysis?

What if we are able to test a vehicle without test riding it? Well apparently its not possible. But its possible for software code. Static code analysis lets you find most generic flaws in your code without executing it.

In a routine testing your code goes through execution with number of concrete inputs and exercise the code in many different ways and try to see what it does, in particular see if it is same as what it is expected. Static analysis takes a different perspective and contrast approach where in the code is examined in a abstract way to to find generic flaws. Static analysis finds out flaws like use of uninitialized memory, possible buffer overruns, memory leaks.

These kind of leak erros may be overlooked in a code review. Not just these static code analysis can also be used to find whether your coding is compliant to development standards like MISRA.

There are lot of static code analysis tools varying by programming languages.

1. FindBugs for java
2. CodeRush for .NET
3. cppcheck and splint for C/C++

- ## How does it work?

Lets take some examples and understand how can automation tools like splint help us out in finguring out some common programming flaws.

List of problems detected by SPLINT in C code: Security vulnerabilities

- Unused declarations
- Type inconsistencies
- Use before definition
- Unreachable code
- Ignored return values
- Execution paths with no return
- Likely infinite loops
- Fall through cases
- Dereferencing a possibly null pointer
- Using possibly undefined storage or returning storage that is not properly defined
- Type mismatches, with greater precision and flexibility than provided by C compilers
- Memory management errors including uses of dangling references and memory leaks
- Dangerous aliasing
- Modifications and global variable uses that are inconsistent with specified interfaces
- Buffer overflow vulnerabilities
- Violations of customized naming conventions

Below is the example code

```
#include <stdio.h>
int main()
{
    char c;
    while (c != 'x');
  {
        c = getchar();
        if (c = 'x')
            return 0;
        switch (c) {
        case '\n':
        case '\r':
            printf("Newline\n");
        default:
            printf("%c",c);
    }
  }
    return 0;
}
```

Splint's output:

Test.c: (in function main)

test.c:5:12: Variable c used before definition An rvalue is used that may not be initialized to a value on some execution path. (Use -usedef to inhibit warning) test.c:5:12: Suspected infinite loop. No value used in loop test (c) is modified by test or loop body.

This appears to be an infinite loop. Nothing in the body of the loop or the loop test modifies the value of the loop test. Perhaps the specification of a function called in the loop body is missing a modification. (Use -infloops to inhibit warning)

test.c:7:9: Assignment of int to char: c = getchar() To make char and int types equivalent, use +charint.

Test.c:8:13: Test expression for if is assignment expression: c = 'x'

The condition test is an assignment expression.

Probably, you mean to use ==

instead of =. If an assignment is intended, add an extra parentheses nesting (e.g., if ((a = b)) ...) to suppress this message. (Use -predassign to

inhibit warning)

test.c:8:13: Test expression for if not boolean, type char: c = 'x' Test expression type is not boolean. (Use -predboolothers to inhibit warning) test.c:14:17: Fall through case (no preceding break) Execution falls through from the previous case (use /*@fallthrough@*/ to mark fallthrough cases). (Use -casebreak to inhibit warning)

Finished checking --- 6 code warnings